Proceeding, Seminar Ilmiah Nasional Komputer dan Sistem Intelijen (KOMMIT 2008)
Auditorium Universitas Gunadarma, Depok, 20-21 Agustus 2008 ISSN : 1411-6286

# PERANCANGAN BAHASA PEMROGRAMAN BERNOTASI PREFIX

[1]Yulia,[2]Rudy Adipranata,[3]Kiswono Prayogo

[1,2,3]Fakultas Teknologi Industri - Jurusan Teknik Informatika
Universitas Kristen Petra Surabaya
[1]yulia@petra.ac.id, [2]rudya@petra.ac.id, [3]kiswono@gmail.com

***ABSTRAK***

*Bahasa pemrograman merupakan alat yang dipergunakan programmer untuk mengimplementasikan algoritma. Salah satu elemen penting dari bahasa pemrograman adalah syntax. Desain syntax yang tidak konsisten dan kurangnya pengetahuan programmer dapat menimbulkan keambiguan dalam penulisan kode. Salah satu cara mengurangi keambiguan adalah dengan konsistensi penggunaan notasi prefix untuk setiap statement.*

*Oleh karena alasan-alasan tersebut, penelitian ini menawarkan bahasa pemrograman baru yang diimplementasikan sebagai interpreter. Tujuan bahasa pemrograman ini adalah untuk membantu programmer mengekspresikan algoritma dalam membuat program sederhana.. Bahasa pemrograman ini dibuat menggunakan C++.*

*Dari hasil pengujian, dapat disimpulkan bahwa bahasa pemrograman ini telah mampu melakukan analisa leksikal dan syntax secara keseluruhan terlebih dahulu, sebelum melakukan proses eksekusi, sehingga dapat mengurangi runtime error.*

*Kata Kunci: Bahasa Pemrograman, Interpreter, Notasi Prefix.*

## 1. PENDAHULUAN

Bahasa pemrograman merupakan alat yang sangat penting bagi *programmer* untuk mengimplementasikan algoritma. Tiap bahasa pemrograman memiliki kelebihan dan kekurangan tersendiri, dan *programmer* memiliki preferensi tersendiri dalam memilih suatu bahasa pemrograman. Beberapa faktor penting seseorang dalam memilih bahasa pemrograman adalah *syntax*, *editor*, dokumentasi, performa, *library*, fleksibilitas, komunitas dan popularitas.

Dalam hal *syntax*, terdapat tiga macam notasi, yaitu *infix*, *prefix* dan *postfix*. Notasi *infix* merupakan notasi yang umum untuk ekspresi matematika, dimana proses atau operator diletakkan diantara subyek dan obyek, contoh bahasa pemrograman yang menggunakan notasi ini adalah C++. Notasi *prefix* adalah notasi yang umum bagi komputer, dimana proses atau operator diletakkan di awal, disertai subyek dan obyek, contoh bahasa pemrograman yang menggunakan notasi ini adalah Lisp.

Kelemahan dari notasi *infix* adalah membutuhkan tanda kurung dan analisis formula dibutuhkan terlebih dahulu untuk menentukan bagian mana yang dikerjakan terlebih dahulu. Kelebihan notasi *prefix* adalah tidak membutuhkan tanda kurung dan kejelasan dalam hirarki, karena *tree* operasi dapat dibentuk dengan mem-*parse* dari sebelah kiri sehingga dapat mempermudah analisis *syntax*.

## 2. TINJAUAN PUSTAKA

### Notasi Sintaks - Prefix

Pemanggilan fungsi biasanya ditulis dengan notasi ini. Dalam notasi ini operator ditulis sebelum *operand*, sehingga dapat memberitahu jumlah dan argumen apa saja yang dibutuhkan, sehingga tidak menimbulkan ambiguitas. Selain itu, tidak dibutuhkan tanda kurung untuk mengevaluasi suatu ekspresi. Kelebihan lain dari notasi ini adalah mudah untuk

dievalusi dengan bantuan *stack*. Notasi ini sering juga disebut *polish notation*.

**Programming Language Creation Process**

**Lexical Analysis**

Analisis leksikal adalah suatu proses mengklasifikasikan urutan karakter tertentu menjadi sebuah token. Token adalah blok karakter yang memiliki kategori. Misalnya token untuk digit bilangan bulat adalah angka berapapun dengan panjang minimal 1 karakter. Analisis leksikal merupakan sub proses dari analisis syntax. Masukan dari proses ini adalah source code dan definisi token, sedangkan keluarannya adalah daftar token.

**Syntax Analysis (Parsing)**

Analisis syntax adalah suatu proses membentuk struktur tree dari token-token yang ada. Syntax adalah deskripsi mengenai urutan token yang valid dalam struktur source code program. Masukan dari proses ini adalah daftar token dan definisi syntax, sedangkan keluarannya adalah parse tree atau syntax tree.

**Code generation**

Pembangkitan kode merupakan proses terpenting dalam pembuatan interpreter atau compiler, pada proses inilah syntax tree diubah menjadi kode yang dapat dieksekusi komputer (Aho, Sethi & Ullrnan, 2001).

**Syntax Representation**

Sebuah sintaks dapat dideskripsikan dalam beberapa cara, melalui teks dengan pengkodean tertentu dengan aturan tertentu ataupun melalui gambar, misalnya dengan *syntax diagram*. Berikut ini penjelasan hal-hal yang berhubungan dengan presentasi suatu sintkas.

**ASCII Character Encoding**

ASCII merupakan singkatan dari American Standard Code for Information Interchange. Pengkodean karakter merupakan salah satu cara untuk mempresentasikan bahasa literal manusia ke dalam komputer. Terdapat banyak pengkodean lain diantaranya yang terkenal adalah Unicode.

**Syntax Diagram**

*Syntax Diagram* merupakan salah satu cara untuk mempresentasikan syntax atau token dalam bentuk gambar. Beberapa cara yang lain untuk mempresentasikan syntax suatu bahasa adalah dengan Backus-Naur Form. Bentuk dari *syntax diagram* ini adalah kotak untuk mempresentasikan suatu elemen, elips untuk mempresentasikan karakter atau kumpulan karakter, serta panah untuk mempresentasikan kemungkinan literal berikutnya.

## 3. METODE PENELITIAN

Adapun metode yang digunakan dalam penelitian ini adalah:

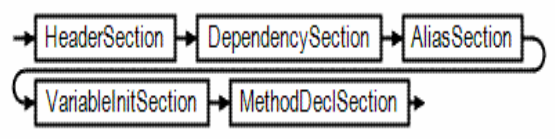
1. Desain struktur file dan elemen-elemen token – menentukan struktur file, elemen-elemen terkecil dalam pembentukan statement, comment dan block sebuah bahasa pemrograman.
2. Desain struktur data dan literal – menentukan struktu data apa saja yang didukung dalam bahasa pemrograman beserta literalnya.
3. Desain syntax – menentukan struktur syntax sebuah statement dan block dalam bahasa pemrograman.
4. Desain tokenizer dan parser – mengimplementasikan desain-desain tersebut di atas sebagai program yang mampu menghasilkan syntax tree.
5. Desain struktur data interpreter – mengimplementasikan penyimpanan data sesungguhnya dalam interpreter.
6. Desain algoritma interpreter – mengimplementasikan fungsi-fungsi yang didukung dan proses eksekusi interpreter.

## 4. PEMBAHASAN

**File Structure Design**

Struktur *file* adalah bagian-bagian yang dibutuhkan dalam membuat *file*

*source code* yang valid. Memiliki struktur sebagai berikut:




Gambar 1. File Structure

*Header Section* merupakan bagian file untuk menandakan tipe dan versi dari *file source code* tersebut.
Contoh:

```
PROGRAM +1
```

Memiliki arti bahwa *file source code* tersebut adalah *file program*, memiliki versi 1.

```
LIBRARY +0.004
```

Memiliki arti bahwa *file source code* tersebut adalah *file library*, memiliki versi 0.004.

*Dependency Section* merupakan bagian *file* untuk menyertakan penggunaan *file* lain di dalam *file* tersebut. *File* terserbut dapat memiliki 2 jenis penggunaan yaitu *require* (relasi dependensi) dan *inherit* (relasi penurunan).
Contoh:

```
INHERIT:RELATIVE "data"
```

Memiliki arti bahwa *file* yang akan digunakan sebagai *parent* turunan terletak relatif terhadap *file* sumber, dan memiliki nama *file* "data.esl".

```
REQUIRE:ABSOLUTE [ "data1" "data2" ]
```

Memiliki arti bahwa *file* yang akan digunakan sebagai dependensi terletak secara absolut dari *file interpreter*, dan memiliki lokasi *file* di dalam *folder* "data1" dan memiliki nama *file* "data2.esl".

*Alias Section* merupakan bagian *file* untuk mengkombinasikan beberapa tipe data menjadi sebuah tipe data, salah satu contohnya adalah *array of string*.
Contoh:

```
VEC1:VectorOfInteger INT32
```

Memiliki arti bahwa tipe data baru bernama "VectorOfInteger" adalah gabungan tipe data *vector* (*array*) yang memiliki tipe *integer* 32-bit.

```
LIST:ListOfString STRING
```

Memiliki arti bahwa tipe data baru bernama "ListOfString" adalah gabungan tipe data *linked list* yang memiliki tipe *string*.
*Variable Initialization Section* merupakan bagian *file* untuk menginisialisasi variabel lokal *file*.
Contoh:

```
VAR:STRING:InitEmpty a
```

Memiliki arti bahwa variabel bernama "a" akan diinisialisasikan sebagai tipe data *string* dengan prosedur inisialisasi "InitEmpty".

```
VAR:INT32:InitValue [ b c d ] +3
```

Memiliki arti bahwa variabel bernama "b", "c" dan "d" akan diinisialisasikan sebagai tipe data *integer* 32-bit dengan prosedur inisialisasi "InitValue" dengan parameter +3.
*Method Declaration Section* merupakan bagian *file* untuk mendeklarasikan sebuah *method* (fungsi atau prosedur).
Contoh:

```
PROC:Main [ ] [ ] [ ]
```

Memiliki arti bahwa prosedur bernama "Main" akan dibuat dengan parameter kosong, variabel lokal prosedur kosong dan tidak memiliki isi.

```
FUNC:STRING:Isi
[
        ]
[       VAR:STRING:InitEmpty [ a
b ]
        ]
[
        STRING:Reset a "isi"
        ]
```

Memiliki arti bahwa fungsi bernama "Isi" akan dibuat dengan parameter kosong, variabel lokal prosedur "a" dan "b", dan memiliki isi mengisi variabel "a" dengan *string* "isi".
Contoh struktur file secara keseluruhan:

```
PROGRAM +1
[       INHERIT:RELATIVE "data"
        REQUIRE:ABSOLUTE        [
"data1" "data2" ]
        ]
[       VEC1:VectorOfInteger
INT32
        ]
[       VAR:INT32:InitValue [ b
c d ] +3
        ]
[       PROC:Main [ ] [ ]
        [       STRING:Reset    a
"nilai baru"
                ]
```

```
]
```

**Syntax Structure Design**

Struktur *syntax* dalam bahasa pemrograman ini menggunakan notasi *prefix* secara menyeluruh, dengan sedikit pengecualian untuk *syntax* yang memiliki parameter dengan jumlah tak tentu. Apabila sebuah sebuah *syntax* memiliki parameter dengan jumlah tidak tentu, maka akan diawali simbol "[" dan diakhiri simbol "]".

*Process Call Statement* merupakan *syntax* untuk memanggil sebuah *method*. Setiap pemanggilan method memiliki awalan berupa tipe data pemilik *method* tersebut. Tanda ":" memiliki arti bahwa *identifier* di sebelah kanan adalah bagian dari *identifier* di sebelah kiri.

Contoh:

```
STRING:ResetConcat a [ "isi" b c ]
```

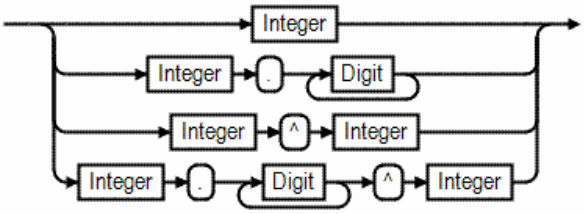
Memiliki arti bahwa prosedur bernama "ResetConcat" merupakan prosedur milik tipe data *string*, proses yang dijalankan adalah menggabungkan *string* "isi" serta nilai variabel "b" dan "c" dan memasukkannya ke dalam variabel "a".

```
INT32:SquareRoot b
```

Memiliki arti bahwa fungsi bernama "SquareRoot" merupakan fungsi milik tipe data *integer* 32-bit, proses yang dijalankan adalah mengembalikan nilai dari akar kuadrat variabel "b".

**Lexical Structure Design**

Struktur leksikal merupakan elemen terkecil dalam pembentukan sebuah token. Sebuah *identifier* harus dimulai dengan huruf, dan boleh diakhiri dengan huruf atau angka. Spasi kosong dan komentar tidak diikutkan dalam prosess *interpreter*. Terdapat 3 jenis komentar, yaitu komentar dokumentasi yang dimulai simbol "{" dan diakhir simbol "}", komentar biasa yang dimulai dengan simbol "(" dan diakhiri simbol ")", dan komentar baris, yang dimulai dengan simbol "~" dan diakhiri penanda akhir baris. *Literal* merupakan *text* yang digunakan *programmer* dalam mendeskripsikan nilai sebuah tipe data dalam *source code*. Terdapat 3 jenis *string literal* yang memiliki arti sama, yaitu menuliskan isi *string* tersebut di dalam pasangan simbol "'", '"', dan "`". Untuk penulisan *integer literal*, tiap angka harus diawali dengan tanda positif atau negatif dan diikuti 1 atau lebih angka. Untuk penulisan numeral literal dijelaskan dalam Gambar 2 berikut ini.




Gambar 2. Numeral Literal

**Data Structure Design**

Terdapat 5 jenis tipe data dalam bahasa pemrograman ini, yaitu:

Basic Data Types adalah tipe data dasar yang dapat dideklarasikan secara langsung. Tipe data ini meliputi: STRING merupakan tipe data yang menyimpan rangkaian karakter. BYTE, INT16, INT32, dan INT64 merupakan tipe data yang menyimpan bilangan integer. Memiliki tambahan berupa huruf U di depan masing-masing tipe sebagai penanda angka positif. FLOAT32 dan FLOAT64 merupakan tipe data yang menyimpan bilangan real. BOOL merupakan tipe data yang menyimpan data biner atau boolean. ANY merupakan tipe data khusus yang tidak dapat dideklarasikan, karena tidak memiliki prosedur inisialisasi, tetapi dapat digunakan untuk penanda struktur kontrol.

Container Data Type adalah struktur data dasar. Tipe data ini tidak dapat dideklarasikan, kecuali dengan menggabungkannya dengan basic data type menjadi sebuah alias data type

atau tipe data baru. Tipe data ini meliputi: VEC merupakan singkatan dari vector, adalah struktur data yang dapat menyimpan nol atau lebih data dengan jenis yang sama. Berbeda dengan array, ukuran vector dapat berubah. Perbedaan VEC1 dan VEC2 terletak pada kecepatan runtime saat perubahan jumlah data, dimana VEC2 cepat pada kedua sisi sedangkan VEC1 hanya cepat pada satu sisi. SET1 dan SET2 merupakan struktur data menyerupai vector, tetapi data yang masuk selalu diurutkan. Angka 1 dalam SET1 memiliki arti single, yang artinya data tidak boleh kembar, sedangkan 2 memiliki arti multi. MAP1 dan MAP2 merupakan struktur data menyerupai gabungan set dan vector, tetapi indeks dari tipe data ini tidak harus berupa bilangan cacah. LIST merupakan struktur data menyerupai vector, tetapi tidak dapat diindeks. Diimplementasikan dengan C++ list.

Extended Data Type adalah tipe data yang dapat dideklarasikan. Merupakan tipe data gabungan yang memiliki variabel dan fungsi di dalamnya.

Alias Data Type adalah tipe data yang dapat dideklarasikan. Merupakan hasil penggabungan tipe data container dengan basic atau extended.

Declarable Data Type meliputi basic data type, extended data type dan alias data type.

## Hasil Pengujian

Bahasa pemrograman ini dapat mengenali 188 jenis error dan 7 jenis warning, misalnya:

```
file open failed, unknown extension, solution
inside solution,  invalid character for identifier
token
eof found before identifier end, too many right
brace dan lain-lain
```

Bahasa pemrograman ini mampu menjalankan 620 method. Tabel berikut ini merupakan perbandingan method yang dimiliki tipe data integer.

Tabel 1.
Perbandingan method yang dimiliki tipe data integer

| Fungsi | UINT64 | UINT32 | UINT16 | UBYTE | INT64 | INT32 | INT16 | BYTE |
|---|---|---|---|---|---|---|---|---|
| AbsOf | | | | | *** | *** | *** | *** |
| Add | *** | *** | *** | *** | *** | *** | *** | *** |
| And | *** | *** | *** | *** | *** | *** | *** | *** |
| CharIsAscii | | | | | | | | *** |
| CharIsControl | | | | | | | | *** |
| CharIsDigit | | | | | | | | *** |
| CharIsGraphical | | | | | | | | *** |
| CharIsHexDigit | | | | | | | | *** |
| CharIsLetter | | | | | | | | *** |
| CharIsLetterOrDigi | | | | | | | | *** |
| CharIsLowercase | | | | | | | | *** |
| CharIsPrintable | | | | | | | | *** |
| CharIsPunctuation | | | | | | | | *** |
| CharIsUppercase | | | | | | | | *** |
| CharIsWhitespace | | | | | | | | *** |
| CharLowercaseOf | | | | | | | | *** |
| CharUppercaseOf | | | | | | | | *** |
| ComplementOf | *** | *** | *** | *** | *** | *** | *** | *** |
| Div | *** | *** | *** | *** | *** | *** | *** | *** |
| Eq | *** | *** | *** | *** | *** | *** | *** | *** |
| Ez | *** | *** | *** | *** | *** | *** | *** | *** |
| Ge | *** | *** | *** | *** | *** | *** | *** | *** |
| Gt | *** | *** | *** | *** | *** | *** | *** | *** |
| Le | *** | *** | *** | *** | *** | *** | *** | *** |
| Lt | *** | *** | *** | *** | *** | *** | *** | *** |
| Mod | *** | *** | *** | *** | *** | *** | *** | *** |
| Mul | *** | *** | *** | *** | *** | *** | *** | *** |
| Ne | *** | *** | *** | *** | *** | *** | *** | *** |
| NegOf | | | | | *** | *** | *** | *** |
| Nz | *** | *** | *** | *** | *** | *** | *** | *** |
| Or | *** | *** | *** | *** | *** | *** | *** | *** |
| Shl | *** | *** | *** | *** | *** | *** | *** | *** |
| Shr | *** | *** | *** | *** | *** | *** | *** | *** |
| Shl | *** | *** | | | | | | |
| Shr | *** | *** | | | | | | |
| Sub | *** | *** | *** | *** | *** | *** | *** | *** |
| ToBaseN | *** | *** | *** | *** | *** | *** | *** | *** |
| ToSTRING | *** | *** | *** | *** | *** | *** | *** | *** |
| Xor | *** | *** | *** | *** | *** | *** | *** | *** |

## Pengujian Analisis Leksikal

Pengujian analisis leksikal dilakukan dengan memberikan *source code* yang secara leksikal tidak lengkap atau salah. Berikut ini adalah contoh *source code* yang salah beserta *output*-nya. Kesalahan terjadi karena terdapat kurung siku kiri yang lebih banyak dari kurung siku kanan. Segmen 1 dan 2 berikut ini adalah *source code* dan hasil interpretasi pengujian ini.

Segmen 1.
Contoh *Source Code Brace Overflow*

```
PROGRAM +1 [ ] [ ] [ ] [ ] [
```

Segmen 2.
*Output Source Code Brace Overflow*

```
 LEXER   ERROR:            E8 too many left brace
 INFO:       1   31     31  5
 PROJECT ERROR:            E14 lexical analyzer
failed, compilation halted
 INFO:   F:/Kiz/Projects/TA/Project/eGaO/Test
Case/pengujian-leksikal-braceoverflow.esp
  Tokenizer Report  Root: 5 / 1  Brace: 5 ( 5 )
```

```
  Size: 11
       1      1    9      9    `IDE`         PROGRAM
       2      1   11     11    `NUM`         1
       3      1   13     13    `[  `        +1
       4      1   15     15    `  ]`        -1
       5      1   17     17    `[  `        +2
       6      1   19     19    `  ]`        -2
       7      1   21     21    `[  `        +3
       8      1   23     23    `  ]`        -3
       9      1   25     25    `[  `        +4
      10      1   27     27    `  ]`        -4
      11      1   29     29    `[  `        +5


  Error Count: 1
```

## Pengujian Analisis *Syntax*

Pengujian analisis *syntax* dilakukan dengan memberikan *source code* yang secara *syntax* tidak lengkap atau salah. Berikut ini adalah contoh *source code* yang salah beserta *output*-nya. Dalam *source code* ini, kesalahan terjadi karena prosedur bernama Concatenate tidak ditemukan pada tipe data STRING, seharusnya menggunakan ConcatOf. Segmen 3 dan 4 berikut ini adalah *source code* dan hasil interpretasi pengujian ini.

Segmen 3.
Contoh *Source Code Procedure Name Not Found*

```
PROGRAM +12.3 [ ] [ ] [ ]
  [
    PROC:Main [ ] [ ]
      [
        STRING:PrintLine
          STRING:Concatenate " a "
        ]
    ]
```

Segmen 4.
*Output Source Code Procedure Name Not Found*

```
 PARSER  ERROR:             E159 function name not
in basic type
 INFO:    Pos:      6   30    117   Type: `IDE`
Data: Concatenate
 PARSER  ERROR:             E83 function parameter
invalid
 INFO:   Main PrintLine
 PROJECT ERROR:             E20 method definition
section syntax analyzer failed, compilation halted
 INFO:   F:/Kiz/Projects/TA/Project/eGaO/Test
Case/pengujian-syntax-procedurenamenotfound.esp

  Error Count: 1
```

## Pengujian *Runtime Error*

Berikut ini adalah contoh *source code* yang salah, karena tidak terdapat prosedur main. Segmen dan 5.20 berikut ini adalah *source code* dan hasil interpretasi pengujian ini.

Segmen 5.
Contoh *Source Code Main Procedure Not Found*

```
PROGRAM +1 [ ] [ ] [ ] [ ]
```

Segmen 6.
*Output Source Code Main Procedure Not Found*

```
EXEC    ERROR:             E182 main procedure not
found
INFO:   F:/Kiz/Projects/TA/Project/eGaO/Test
Case/test2.esp
PROJECT ERROR:             E181 interpretation
failed
INFO:   F:/Kiz/Projects/TA/Project/eGaO/Test
Case/test2.esp

 Error Count: 1
```

## Evaluasi

- Tingkat *readability* relatif menengah berdasarkan hasil kuisioner dengan rata-rata pemahaman 58,91 %
- Faktor *writability* bahasa pemrograman ini relatif rendah, karena bentuk *syntax* yang kurang fleksibel dibandingkan C++
- *Reliability* bahasa pemrograman ini tergolong rendah
- Faktor *writability* bahasa pemrograman ini relatif rendah, karena bentuk *syntax* yang kurang fleksibel dibandingkan C++

## 5. KESIMPULAN

Bahasa pemrograman yang dibuat dalam penelitian ini tidak menggunakan sistem *read-eval-print*, sehingga mengurangi jumlah *runtime error* dan proses eksekusi yang tidak perlu.

Bahasa pemrograman ini telah mampu melakukan analisa leksikal dan syntax secara keseluruhan terlebih dahulu, sebelum melakukan proses eksekusi, sehingga dapat mengurangi *runtime error*.

Bahasa pemrograman ini memiliki *readability* relatif menengah, *writability* dan *reliability* rendah, dan *cost* yang tinggi.

Diharapkan bahasa pemrograman ini dapat dikembangkan lebih lanjut dengan desain yang lebih baik serta menggabungkannya ke dalam sebuah *integrated development environment.*

## DAFTAR PUSTAKA

[1] Aho, Alfred V., Sethi, Ravi and Ullrnan, J. D. 2001. *Compilers: Principles, techniques, and tools.* Massachussetts: Addison-Wesley.

[2] *Introduction to programming using Python*. 2008. 25 Juni 2008, http://www.pasteur.fr/formation/infobio/python/ch05s02.html

[3] Jinks, Pete. (n.d.). *Notations for context-free grammars*. 28 Mei 2008, http://www.cs.man.ac.uk/~pjj/bnf/bnf.html

[4] Joung , Yuh-Jzer. (n.d.). *Syntatic structure*. 28 Mei 2008, http://joung.im.ntu.edu.tw/teaching/pl/2001/syntatic_structure.pdf

[5] Kayabasi, Alp. (n.d.). *Prefix notation*. 28 Mei 2008, http://triton.towson.edu/~akayabas/COSC455_Spring2000/Prefix.html

[6] Sebesta, R. W. 2006. *Concepts of programming language (7th edition)*. Boston: Addison-Wesley.

[7] Stroustrup, Bjarne. (n.d.). *C++ programming language*. 28 Mei 2008, http://www.research.att.com/~bs/C++.html

[8] *TJ CompSci*. (n.d.). 28 Mei 2008, http://academics.tjhsst.edu/compsci/CS2C/U3/syndiag.html